University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday at 11:59 pm
  - Lab 1: next Monday at 11:59 pm
- Assignment 1 hasn't been posted yet (sorry for the delay)
- TAs student support hours available on the syllabus page

Previous lecture

- ADT Tree
  - Examples and basic definitions
  - TreeInterface

Muddiest Points

- When should I use recursion versus some other data structure ?
- Why the root node is not traversed
- how do you distinguish whether a child node on a tree is left or right
- Just the concept of binary search trees in general, will have to revisit them independently at a later time.
- what does it mean for a set of data to be dynamic?
- Why the hash function's runtime is an average and the others are not
- full vs complete tree. Are the left are right sides arbitrary or do they matter what side?
- non linear structures. Is it just trees or do graphs count?
- what a node is used for

Muddiest Points

- what a node is used for
- I'm confused why we need to analyze the search structure.
- We talked about hashing (as a future topic). I understand using the mod function to assign to a position in the hash table, but I never understood what comes after that. I'm looking forward to having this explained in more detail later in the semester.
- What is the recurrence relation for boggle board BF
- Is it that simple that a complete tree is filled left to right?
- why unsorted linkedlist can use binary search
- in the binary tree graphic, is the rightmost tree a binary tree or not?
- What does ADT have to do with trees?

Tree Implementation: Code Walkthrough

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

buildTree method

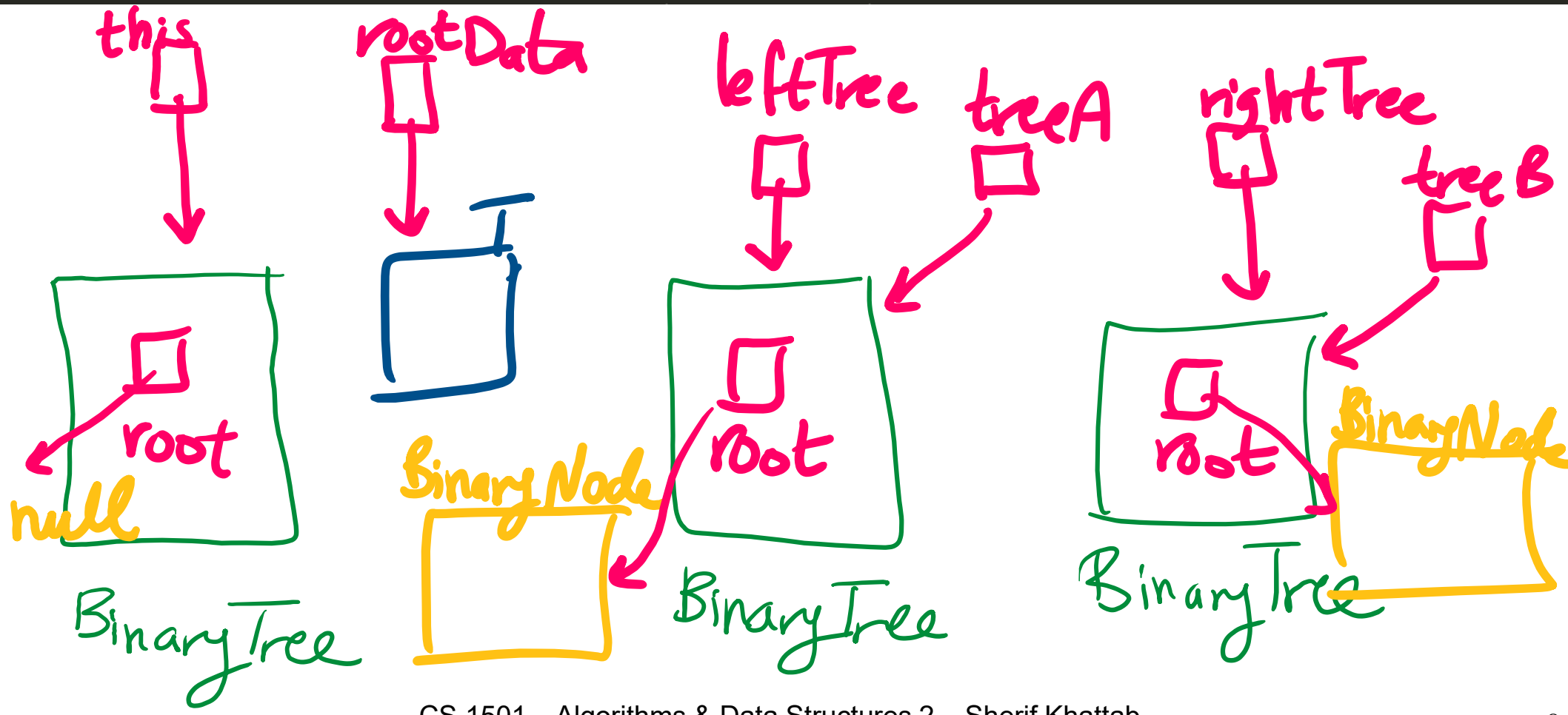
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```

Let's draw a picture of the before state

- Given the call

```
privateBuildTree(data, treeA, treeB);
```

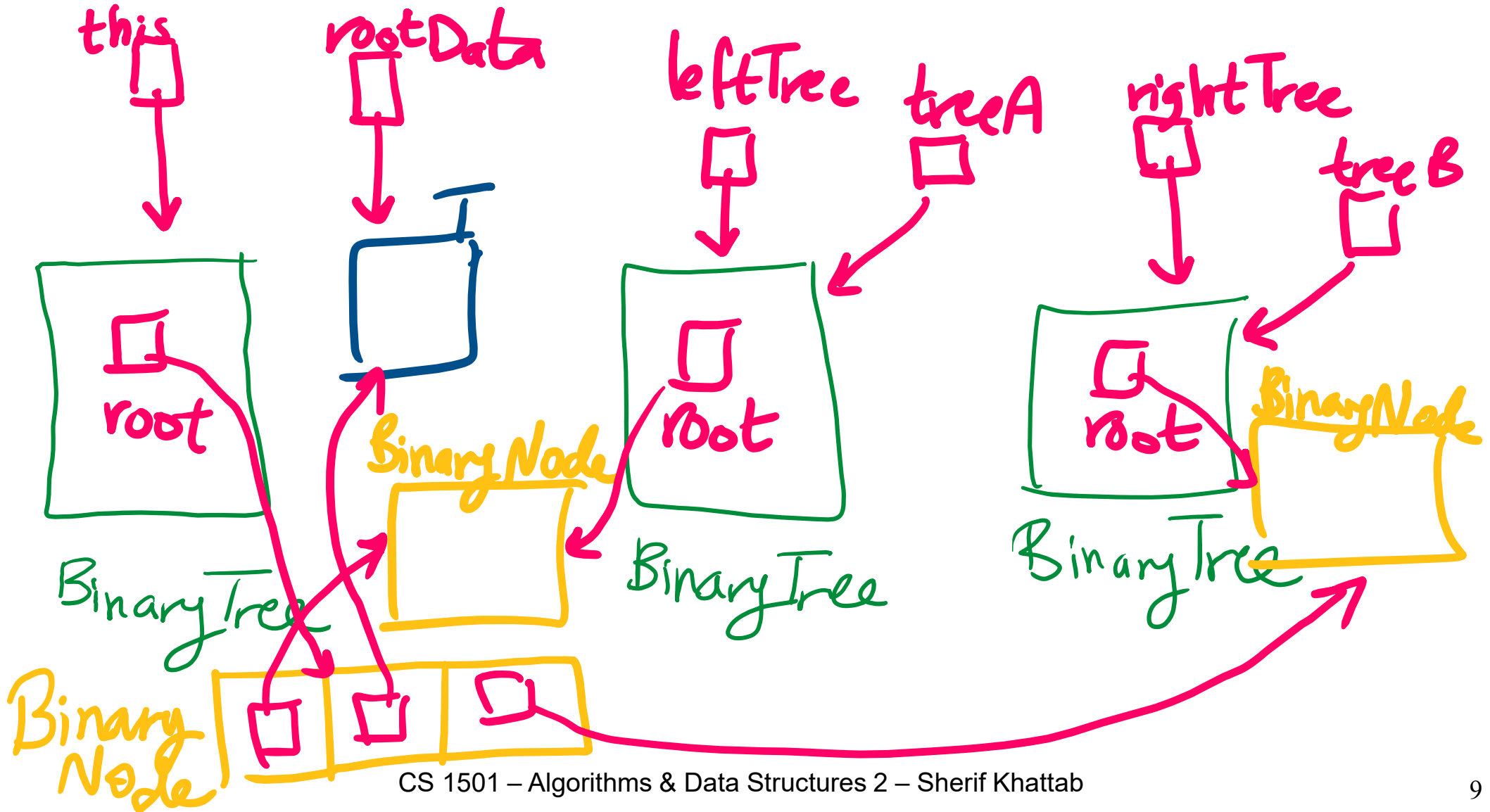
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```



Let's draw a picture of the after state

```
privateBuildTree(data, treeA, treeB);
```

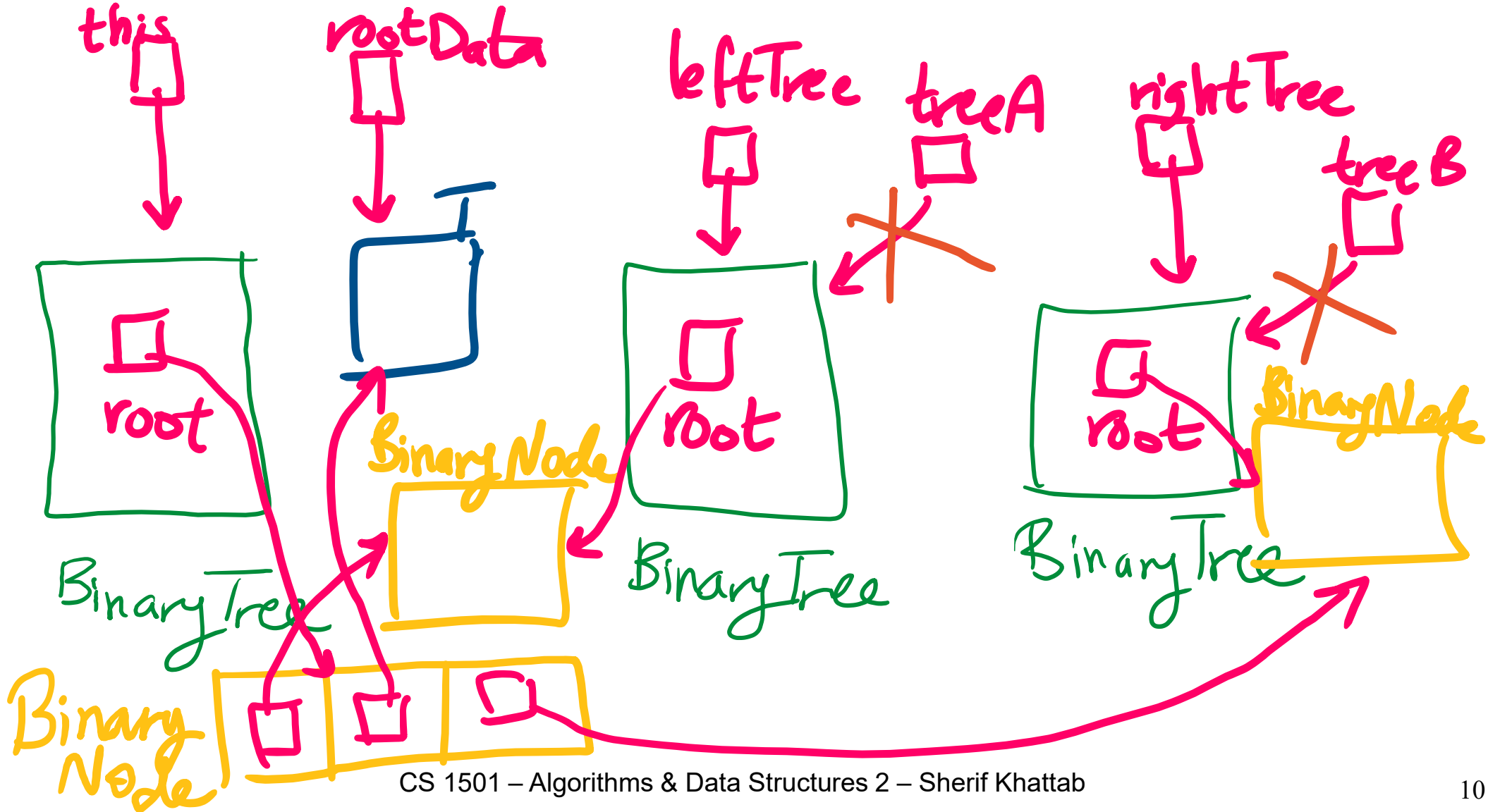
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```



Let's draw a picture of the after state

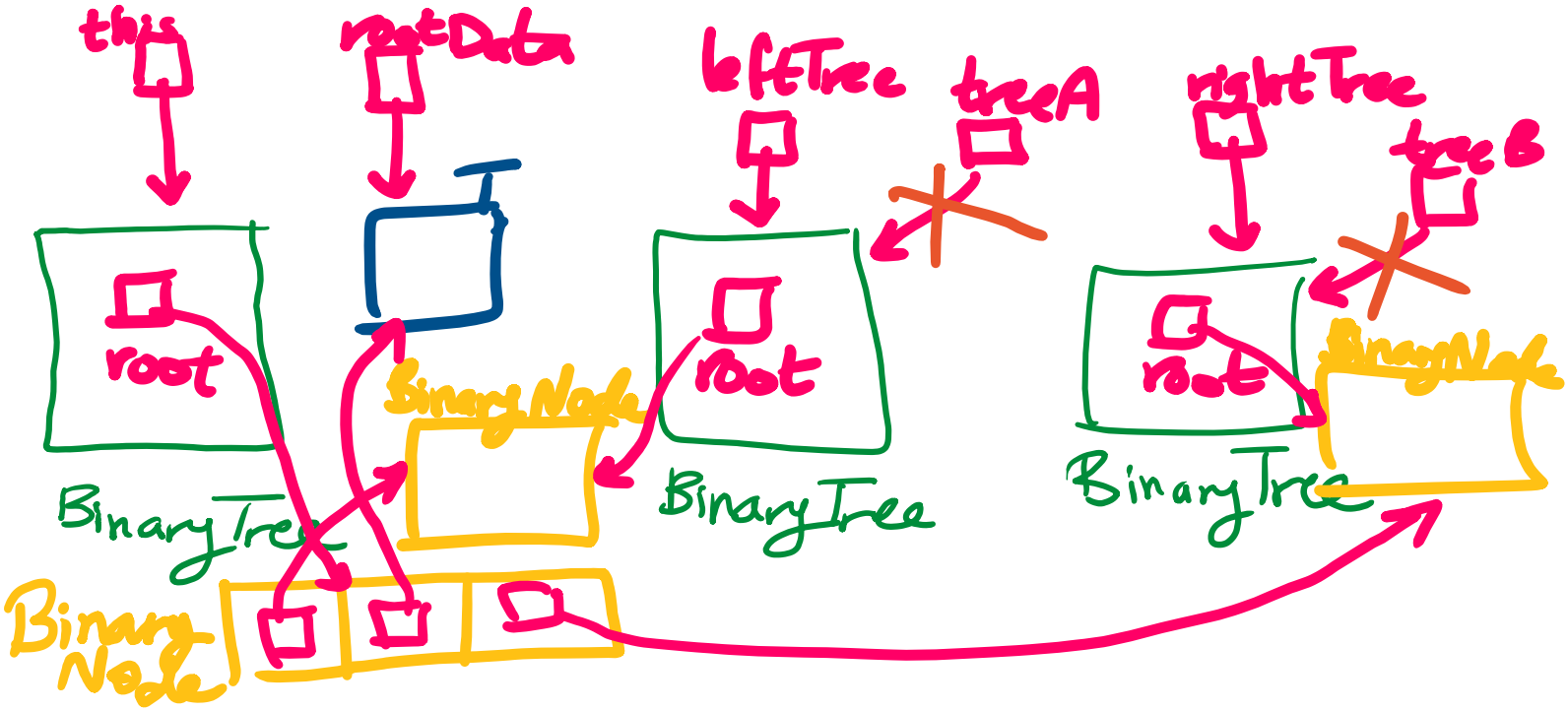
Need to also Prevent client direct access to this

`treeA` shouldn't have access this.root.left (same for treeB)



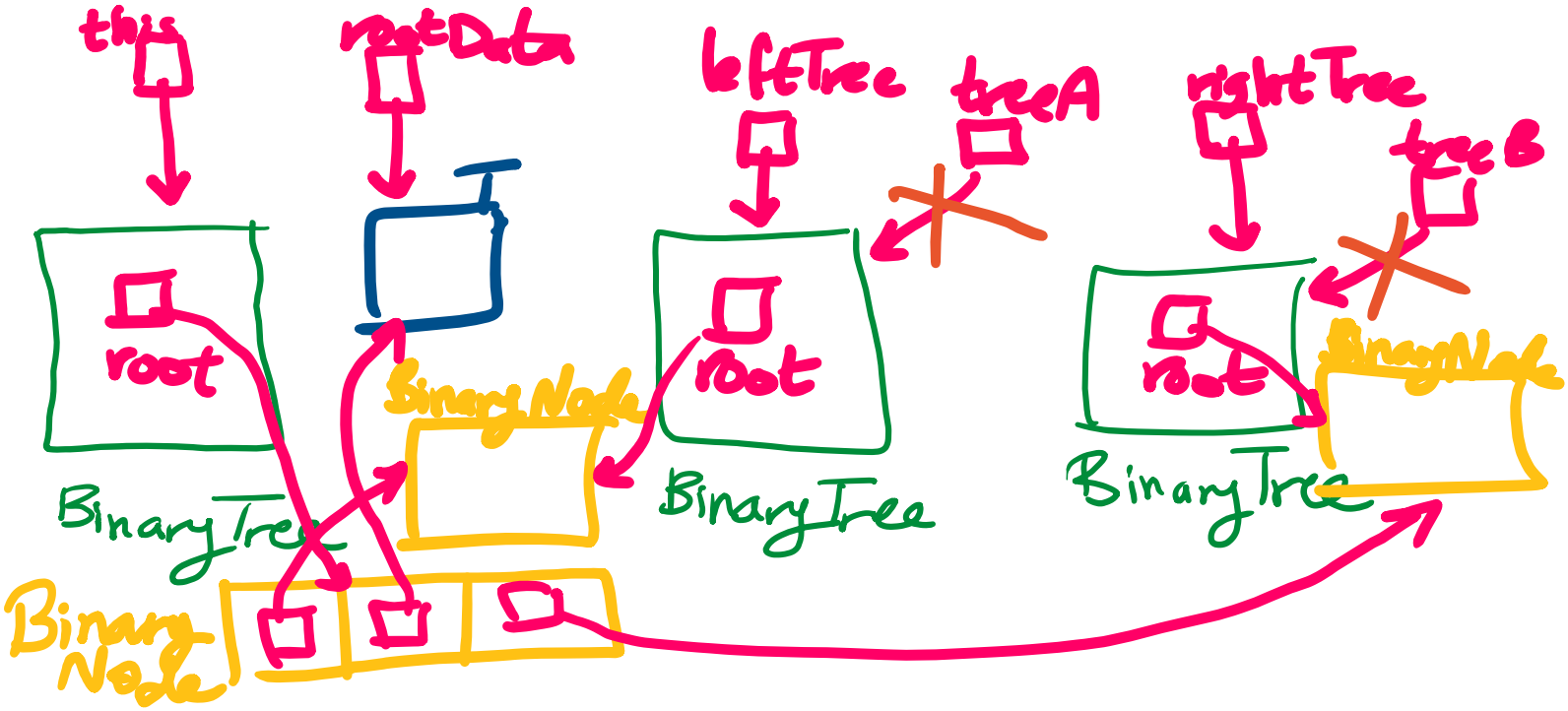
Main logic

- `root = new BinaryNode<>(rootData);`
- `root.left = leftTree.root;`
- `root.right = rightTree.root;`
- How to prevent client access?



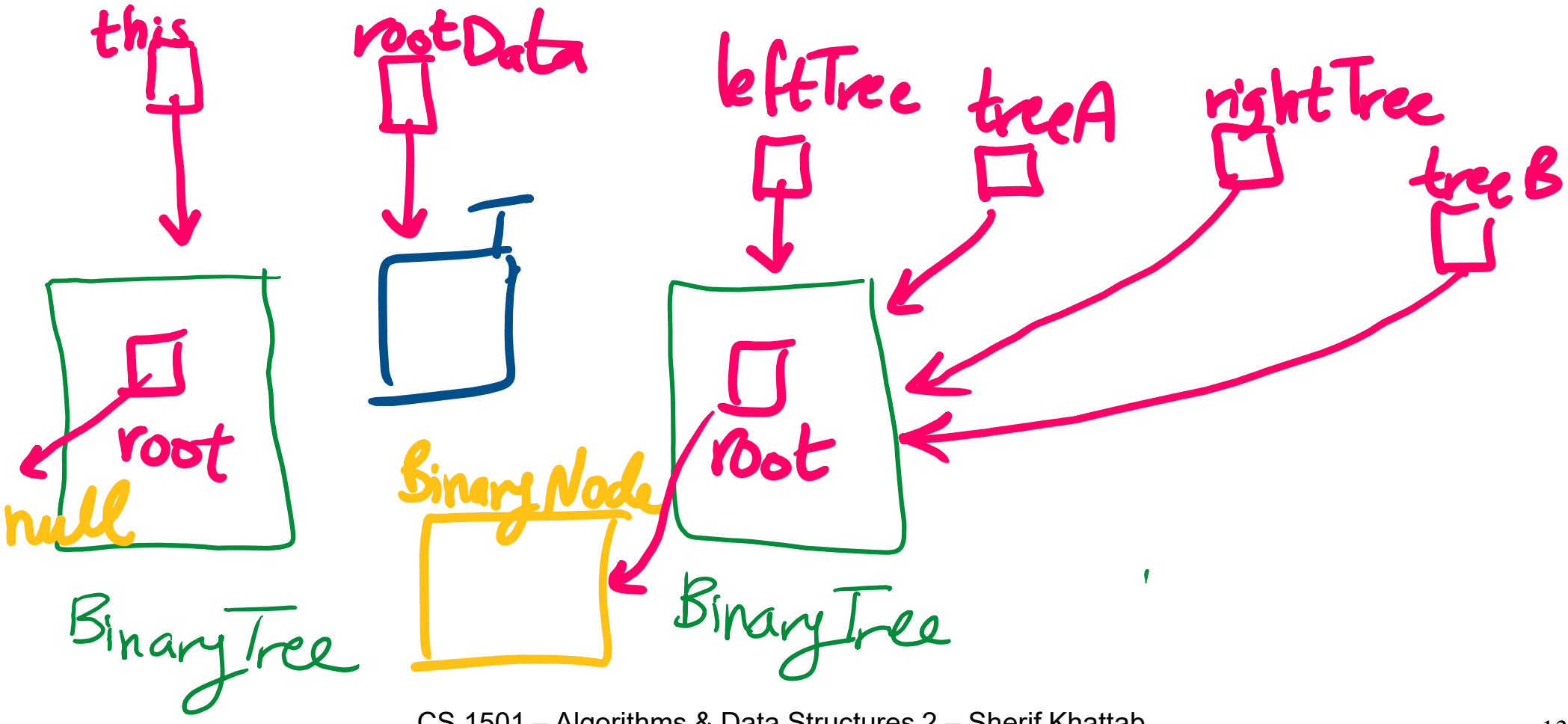
How to prevent client access?

- treeA = treeB = null; //is that possible?
- leftTree = rightTree = null; //would that work?
- leftTree.root = null; rightTree.root = null;



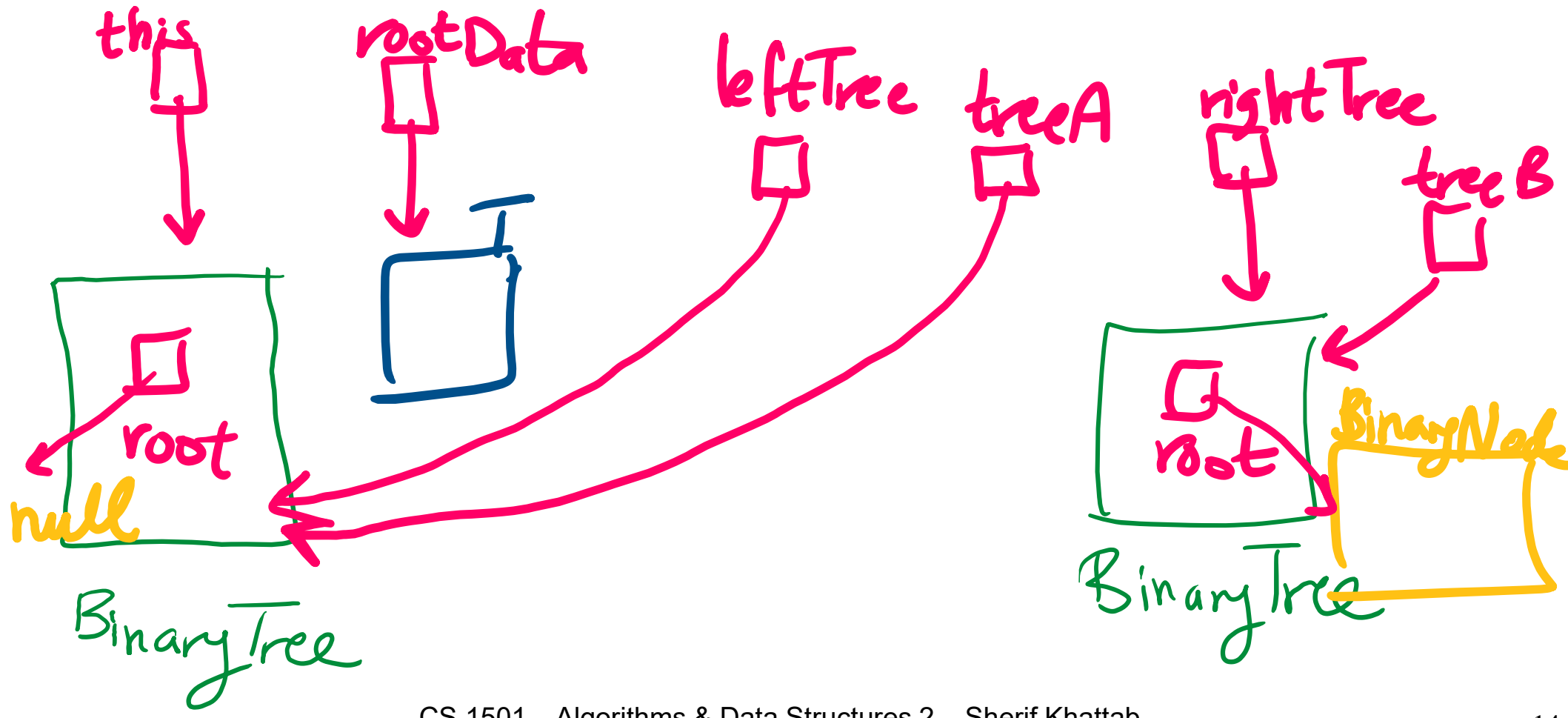
Special case: treeA == treeB

Need to make a copy of leftTree.root



Special case: treeA == this or treeB == this

Need to be careful before leftTree.root = null and rightTree.root = null



Tree Traversal Methods

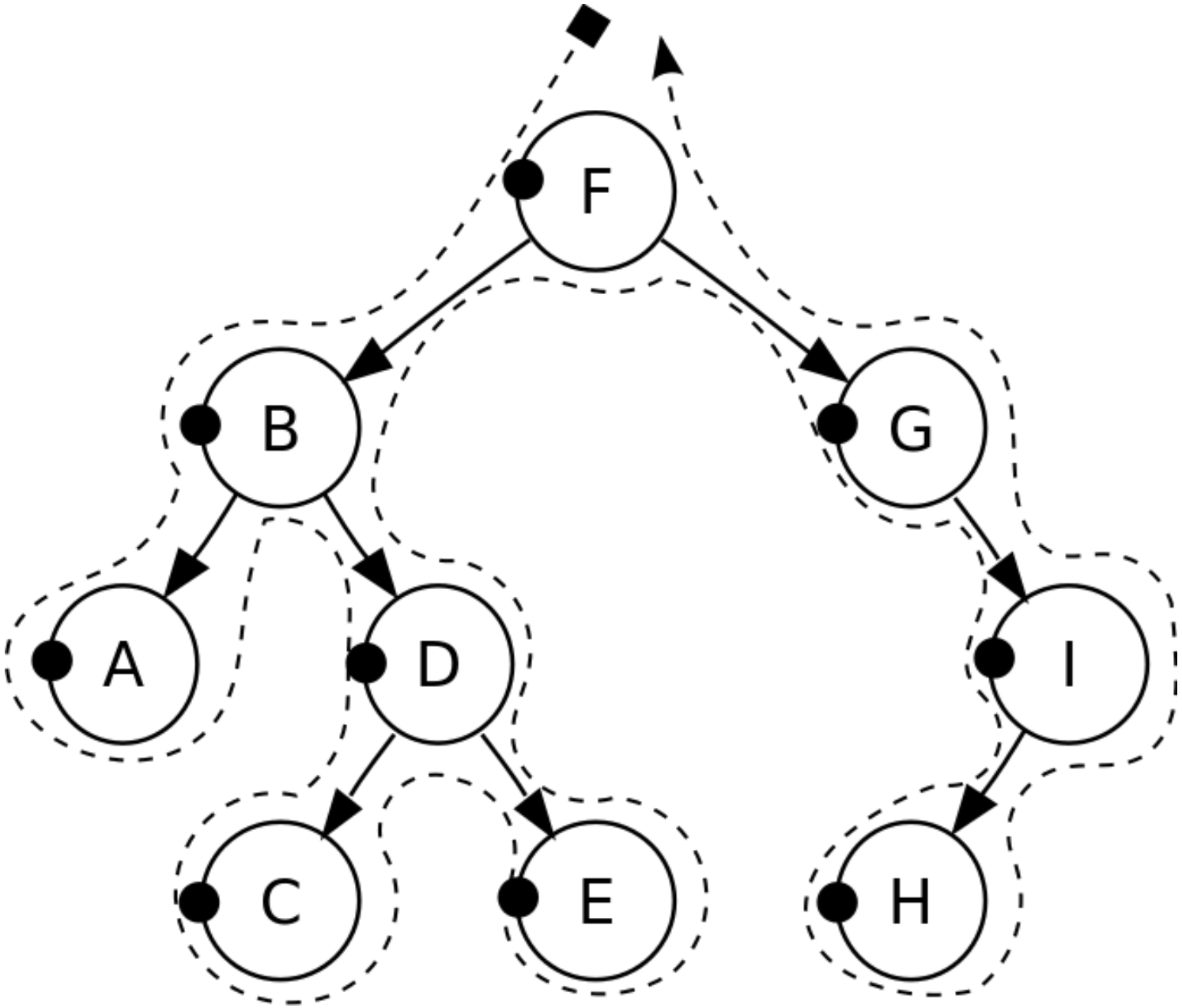
- How to traverse a Binary Tree
  - General Binary Tree
    - Pre-order, in-order, post-order, level-order

Traversals of a General Binary Tree

- Preorder traversal
  - Visit root **before** we visit root's subtree(s)

Pre-order traversal

- F
- B
- A
- D
- C
- E
- G
- I
- H



Pre-order traversal implementation

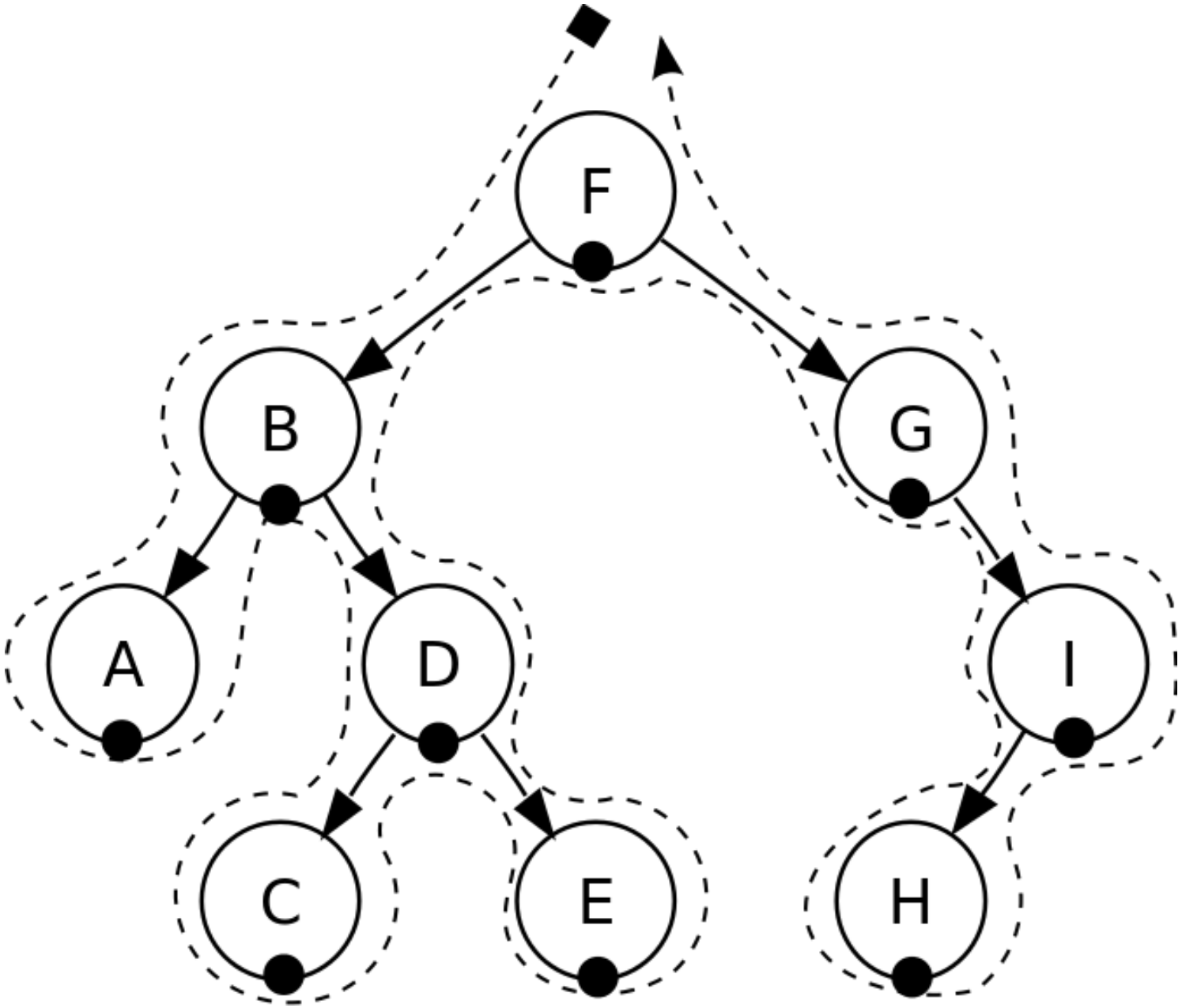
```
void traverse(BinaryNode<T> root){

   if(root != null){

      System.out.println(root.data);

      traverse(root.left);

      traverse(root.right);

    }

}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- In-order traversal
  - Visit root of a binary tree **between** visiting nodes in root's subtrees.
  - left then root then right

In-order traversal

A
B
C
D
E
F
G
H
I



In-order traversal implementation

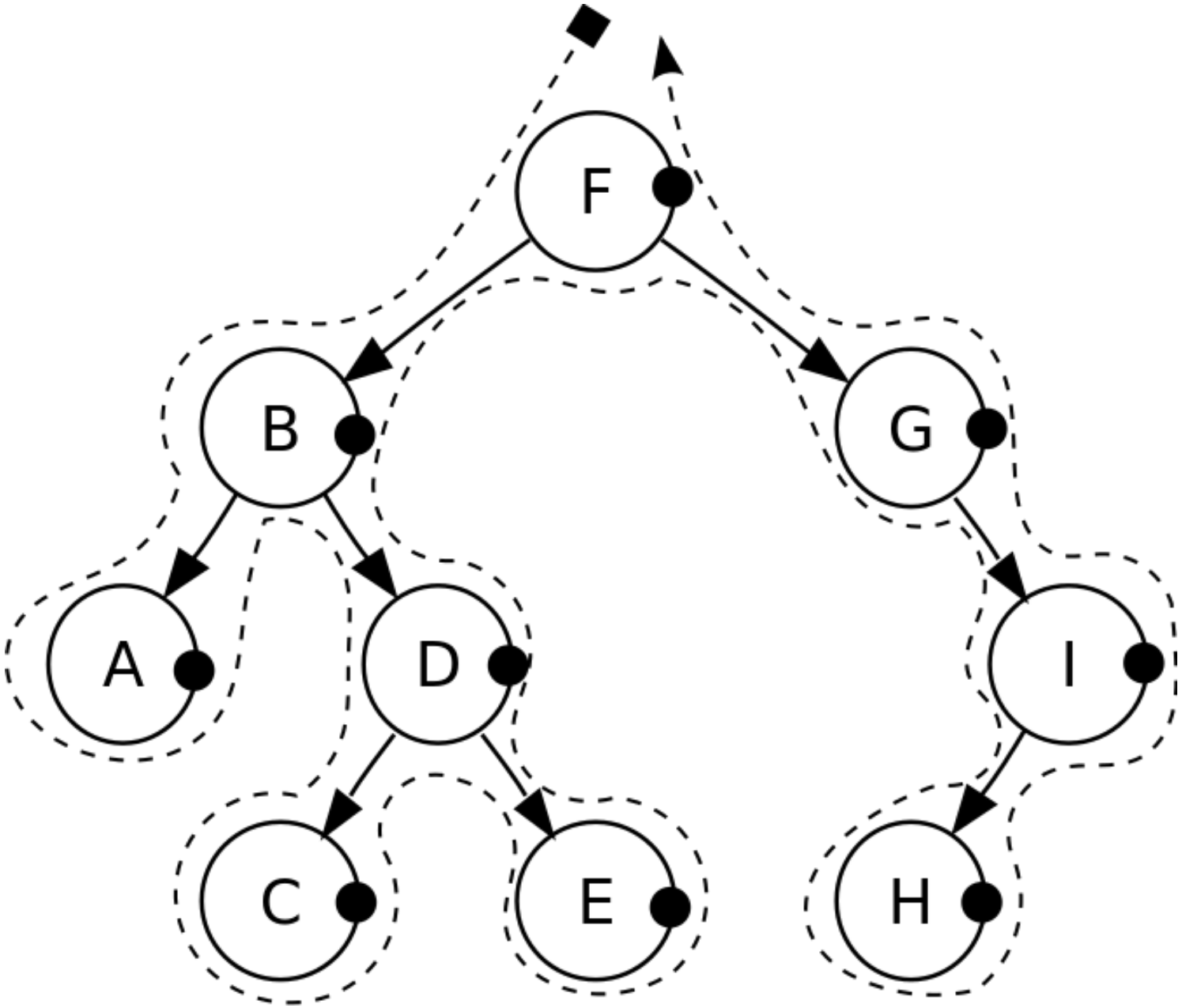
```
void traverse(BinaryNode<T> root){

   if(root != null){

      traverse(root.left);

      System.out.println(root.data);

      traverse(root.right);

    }

}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- Inorder traversal
  - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
  - Visit root of a binary tree after visiting nodes in root's subtrees

Post-order traversal

A
C
E
D
B
H
I
G
F



Post-order traversal implementation

```
void traverse(BinaryNode<T> root){

   if(root != null){

      traverse(root.left);

      traverse(root.right);

      System.out.println(root.data);

    }

}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- Inorder traversal
  - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
  - Visit root of a binary tree after visiting nodes in root's subtrees
- Level-order traversal
  - Begin at root and visit nodes one level at a time
  - We will see the implementation when we learn Breadth-First Search of Graphs

Tree Search Take 1

- *Traverse* every node of the tree
  - Is the key inside the node equal to the target *key*?
- How can we traverse the tree?

Tree Search Take 1

What is the runtime?

Can we do better?

Can we traverse the tree more intelligently?

Tree Search Take 2: Binary Search Tree

- Search Tree Property
  - left.data < root.data < right.data
  - Holds for each subtree
  - In Java:
    - root.data.compareTo(left.data) > 0 &&
    - root.data.compareTo(right.data) < 0